

Symbolic BDD and ADD Algorithms for Energy Games

Shahar Maoz Or Pistiner Jan Oliver Ringert

School of Computer Science
Tel Aviv University, Israel

Energy games, which model quantitative consumption of a limited resource, e.g., time or energy, play a central role in quantitative models for reactive systems. Reactive synthesis constructs a controller which satisfies a given specification, if one exists. For energy games a synthesized controller ensures to satisfy not only the safety constraints of the specification but also the quantitative constraints expressed in the energy game. A symbolic algorithm for energy games, recently presented by Chatterjee et al. [10], is symbolic in its representation of quantitative values but concrete in the representation of game states and transitions. In this paper we present an algorithm that is symbolic both in the quantitative values and in the underlying game representation. We have implemented our algorithm using two different symbolic representations for reactive games, Binary Decision Diagrams (BDD) and Algebraic Decision Diagrams (ADD). We investigate the commonalities and differences of the two implementations and compare their running times on specifications of energy games.

1 Introduction

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from its temporal logic specification [19]. Rather than manually constructing a system and using model checking to verify its compliance with its specification, synthesis offers an approach where a correct implementation of the system is automatically obtained, if such an implementation exists. Traditionally, specifications for synthesis express qualitative properties of desired system behavior, which are either satisfied or not satisfied. Over the last decade interest has increased for quantitative properties [2], which can express, e.g., cost of actions or probability of success, and allow for synthesis of optimal solutions.

One intuitive and popular quantitative extension of games for reactive synthesis are energy games (EGs), defined by Bouyer et al. [6], which model quantitative consumption of a limited resource, e.g., cost, time, or energy. Transitions between states are annotated with weights. A play starts with a finite energy level that is updated by the weight on every transition. An infinite play is winning for the system if the energy level never goes below 0. The EG is realizable if the system has a strategy to win for initial choices of the environment. The objective of synthesis for EG is to construct such a strategy.

Brim et al. [7] have presented an efficient pseudo-polynomial algorithm for solving EGs (polynomial in the state space and the maximal weight). The algorithm is bounded by the maximal initial energy. Because the maximal initial energy for realizable energy games is bounded the algorithm is complete. It computes the minimal energy required to win from any state in a backwards manner, using a fixed point calculation. A representation by minimal energy levels is symbolic in the sense of antichains [12], also used in the algorithm of Chatterjee et al. [10]. These algorithms are thus symbolic in the quantitative values but concrete in the representation of game states and transitions¹.

In this work we present a novel algorithm that is optimized for reactive energy games and symbolic both in the quantitative values and in the underlying game representation. Our algorithm

¹Note that the implementation presented by Chatterjee et al. [10] avoids a full concretization of underlying safety games by using antichains as described in [14].

implements a fixed point computation similar to Chatterjee et al. [10] but due to the reactive nature of the game it updates energy levels of system and environment states in one instead of two steps. Our symbolic algorithm also uses antichains. However, the antichains are now defined over sets of states and their sets of energy levels instead of over single states and their sets of energy levels. These modifications require efficient symbolic data structures and corresponding operations to represent the (intermediate) results of the fixed point computation.

We further present two different implementations of our algorithm. The first implementation is based on Binary Decision Diagrams (BDDs) [8] to represent states, transitions, and weight definitions. Antichains are then maps mapping minimal energy levels to BDDs. Accordingly, the fixed point calculation of the algorithm is based on iterations over minimal energy levels and weights and is executed using symbolic BDD operations.

Our second implementation is based on Algebraic Decision Diagrams (ADDs) [1], a special case of Multi-Terminal Binary Decision Diagrams (MTBDDs) [15]. ADDs have numbers as terminal nodes — instead of TRUE and FALSE in the case of BDDs — and provide efficient implementations of symbolic algebraic operations. With ADDs, an antichain in our algorithm is expressed in a single ADD. Again, we have implemented the fixed point calculation of the algorithm using only symbolic ADD operations.

The ADD and BDD algorithms do not only use symbolic data structures for efficient representation: **contributions of our implementations are also their specific use of symbolic manipulations for efficiently performing quantitative operations.** We explain both implementations to highlight their commonalities and differences. In Sect. 7 we compare the running times of the BDD and ADD implementations against each other and against themselves over increasing sizes of specifications and different specifications of weights.

We present background on games, BDDs, ADDs, and EGs in Sect. 2. Sect. 3 introduces a running example specification. We present our generic algorithm in Sect. 4, and its BDD and ADD implementations in Sect. 5 and Sect. 6 resp. Sect. 7 presents a preliminary evaluation of our two implementations. We discuss the results and related work in Sect. 8 and conclude in Sect. 9.

2 Preliminaries

2.1 Infinite Games, BDDs, ADDs

We repeat some basic definitions of games, BDDs, and ADDs. We also describe the general approach for symbolically representing games between an environment and a system player using BDDs.

Games, game graphs, plays, and strategies We consider infinite games played between two players on a finite directed graph as they move along its edges. For a game we define a *game graph* as a tuple $\Gamma = \langle G = \langle V, E, w \rangle, V_0, V_1 \rangle$, where $G = \langle V, E, w \rangle$ is a finite directed graph with a weight function $w : E \rightarrow \mathbb{Z}$ that attaches weights to its edges. V_0, V_1 is a partition of V into V_0 , the set of player-0 (the maximizer) vertices, and V_1 , the set of player-1 (the minimizer) vertices. Each vertex $v \in V$ has out degree at least one, i.e., G has no deadlocks. A play starts by placing a pebble on a given initial vertex, and continues infinitely many rounds as the two players move the pebble along the edges of G . In each round, if the pebble is on a vertex $v \in V_i$, $i \in \{0, 1\}$, then player- i chooses an outgoing edge from v to some adjacent vertex $u \in V$, and the next round starts with the pebble on u . The infinite path formed by the rounds is called a *play*. A *strategy* of player- i is a function that given the prefix of a play ending in a vertex of player- i returns a successor vertex.

Binary Decision Diagrams and Algebraic Decision Diagrams Binary decision diagrams (BDDs) [8] are a compact data structure for representing and manipulating Boolean functions. For the purpose of verification and synthesis they are usually used to represent sets of Boolean vectors [16], e.g., encoding sets of states or transitions. Bryant [8] showed how Boolean operations on BDDs can be efficiently implemented, including logical connectives and existential and universal abstractions. ADDs [1] are a generalization of BDDs, such that the terminal nodes may take on values belonging to a set of constants different from 0 and 1, such as integers or reals. Another name for ADDs is Multi-Terminal Binary Decision Diagrams (MTBDDs) [15], which reflects their structure rather than their application for computations in algebras. An ADD represents a Boolean function of n variables $f : \{0, 1\}^n \rightarrow S$ where S is a set of constants. Boolean, arithmetic, and abstraction operations are all applicable to ADDs. We now briefly state the most important operations that we use in our ADD based algorithm. For ADDs g, h , and a 0-1 ADD f , arithmetic operators $\text{op} \in \{+, -, \cdot, \max, \dots\}$, denoted $g \text{ op } h$, operate on the terminal nodes for common variable assignments, the *If-then-else* Boolean operation is defined as $\text{ITE}(f, g, h) = f \cdot g + \neg f \cdot h$, and abstraction of variables v from g , which aggregate terminal values of g for all assignments to variables in v by operators *min* or *max*.

Symbolic representation of game graphs for reactive games Reactive games are turn-based two players games, between an environment and a system player, in which the environment always plays first and the system *reacts*. The environment controls the input variables, denoted by *env*, and the system controls the output variables, denoted by *sys*, all are assumed to be Boolean, such that their values are modified in each step of the game [19]. We denote by $\mathcal{V}_{var} := \{0, 1\}^{var}$ all the possible assignments to the variables *var*, where $s_{var} \in \mathcal{V}_{var}$ is some assignment to *var*. We now define the symbolic weighted game graph $G^{sym} := \langle \theta^e, \theta^s, \rho^e, \rho^s, w \rangle$ for reactive games. A *state* s in G^{sym} is an assignment to *all* variables, i.e., $s := s_{env \cup sys} \in \mathcal{V}_{env \cup sys}$. That means each state belongs to both players (unlike the general model). We denote by θ^e, θ^s the sets of *initial* states of the environment and the system, respectively. Each is represented by a BDD (or by a 0-1 ADD) that encodes its characteristic function: $\chi_{\theta^e} : \mathcal{V}_{env} \rightarrow \{0, 1\}$ and $\chi_{\theta^s} : \mathcal{V}_{env \cup sys} \rightarrow \{0, 1\}$. Each player has a *transition relation* that defines its valid transitions in G^{sym} . It is denoted by ρ^e and ρ^s for the environment and the system, respectively. A next state in G^{sym} , as opposed to a current state, is represented by a *primed* version of the variables sys' and env' . Therefore, the BDDs (or 0-1 ADDs) that represent ρ^e and ρ^s encode the characteristic functions $\chi_{\rho^e} : \mathcal{V}_{env \cup sys \cup env'} \rightarrow \{0, 1\}$ and $\chi_{\rho^s} : \mathcal{V}_{env \cup sys \cup env' \cup sys'} \rightarrow \{0, 1\}$. The bijection *prime* : $\mathcal{V}_{sys \cup env} \rightarrow \mathcal{V}_{sys' \cup env'}$ replaces unprimed variables by their primed counterparts. We denote a transition from state s_1 to state s_2 by $t_{s_1, s_2} := s_1 \cup \text{prime}(s_2) \in \mathcal{V}_{env \cup sys \cup env' \cup sys'}$. It is valid if $t_{s_1, s_2} \in \rho^e \cap \rho^s$, thus consists of a valid transition for environment choice $s_2^e := s_{env} \in \mathcal{V}_{env}$, denoted by $t_{s_1, s_2^e} := s_1 \cup \text{prime}(s_2^e) \in \rho^e$, followed by a valid transition for system choice $s_2^s := s_{sys} \in \mathcal{V}_{sys}$, denoted by $t_{s_1, s_2^e, s_2^s} := s_1 \cup \text{prime}(s_2^e) \cup \text{prime}(s_2^s) \in \rho^s$. Also, G^{sym} has a *weight function* $w : \mathcal{V}_{env \cup sys} \times \mathcal{V}_{env \cup sys} \rightarrow \mathbb{Z} \cup \{\perp\}$ that attaches weights to its transitions, such that for all pairs of states s_1, s_2 if $t_{s_1, s_2} \in \rho^e \cap \rho^s$, $w(s_1, s_2) \in \mathbb{Z}$, and otherwise, $w(s_1, s_2) = \perp$. For details of its actual representation see Sect. 5 and Sect. 6.

Notation We work with abstractions where a BDD represents a set of states or transitions, i.e., $BDD \equiv \text{Set of States or } BDD \equiv \text{Set of Transitions}$. An ADD represents a function assigning an integer, plus or minus infinity, to every state or transition, i.e., $ADD \equiv \text{States} \rightarrow \mathbb{Z}_{\pm\infty}$ or $ADD \equiv \text{Transitions} \rightarrow \mathbb{Z}_{\pm\infty}$.

2.2 Energy Games

Energy games (EG), also called the *lower bound problem*, have been studied by Bouyer et al. [6]. We give the formal definitions of EG as defined in [6, 7], followed by a fixed point formulation of the solution for EG as in [6, 10].

EG definition and objectives We add to an infinite play on an EG graph Γ an *initial credit* or *initial energy* value $c \in \mathbb{N}$. We define the *energy level* of the prefix $v_0v_1 \dots v_j$ of the play $p = v_0v_1 \dots v_j \dots$, by $\text{EL}(p, j) = c + \sum_{i=0}^{j-1} w(v_i, v_{i+1})$. The objective of player-0 is to construct an infinite play $p = v_0v_1 \dots v_j \dots$ such that the energy level is always non-negative during the play, i.e., $\text{EL}(p, j) \geq 0$ for all $j \geq 1$. We say that such a play p is *winning* for player-0. Otherwise, if the energy level goes below zero during p , it is winning for player-1. Given an initial credit c , a strategy is *winning* for player-0 from $v \in V$ for c if all plays resulting from the strategy are winning for player-0. A vertex v is winning for player- i if there exists a winning strategy for player- i from v for some initial credit c .

We consider two EG problems: (1) *Decision problem*: Given a vertex v , the problem asks if there exists an initial amount of energy that suffices for player-0 to win from v ; and (2) *Minimum credit problem*: which asks for every vertex $v \in V$, what is the *minimal* initial amount of energy that suffices for player-0 to win from v . A solution of the second also provides a solution for the first by checking whether the minimal initial energy is finite. In addition a solution for the second allows for the construction of optimal memoryless strategies for solving EGs [10]. We thus focus on the minimum credit problem.

Algorithm for solving energy games from [10] Both Chatterjee et al. [10] and Bouyer et al. [6] present algorithms to solve the minimal credit problem of EGs. Their solutions are presented as the greatest fixed point of a monotone operator on a power set lattice. Given an EG graph $\Gamma = \langle G = \langle \text{States}, E, w \rangle, \text{States}_0, \text{States}_1 \rangle$ and a bound $c \in \mathbb{N}$, we denote by $SE(c)$ the pairs of states and energy values up to c , i.e., $\text{States} \times \{n \in \mathbb{N} \mid n \leq c\}$. We relate to the monotone bounded operator $\text{Cpre}_c : \mathbb{P}(SE(c)) \rightarrow \mathbb{P}(SE(c))$, where $\mathbb{P}(SE(c))$ is the power set of $SE(c)$, that is defined in [10, Eqn. 1]. Chatterjee et al. [10] present a symbolic fixed point algorithm that computes a set of pairs consisting of a state and an energy level that suffices for player-0 to win for that state. It performs iterative applications of Cpre_c to $SE(c)$, which results in a finite \subseteq -descending chain whose last element approximates the greatest fixed point of Cpre_c .

Symbolic representation of upward closed sets We define a *partial order* $\preceq \subseteq SE(c) \times SE(c)$ (i.e., reflexive, transitive, antisymmetric binary relation), such that for all $(s, e), (s', e') \in SE(c)$, $(s, e) \preceq (s', e')$ iff $s = s' \wedge e \leq e'$. We call $\text{Up}(\preceq, A) := \{(s, e) \in SE(c) \mid \exists (s', e') \in A : (s', e') \preceq (s, e)\}$ the \preceq -*upward closure* of $A \subseteq SE(c)$. A set $A \subseteq SE(c)$ is \preceq -*upward-closed* if for all $(s, e), (s', e') \in SE(c)$, if $(s, e) \in A$ and $(s, e) \preceq (s', e')$, then $(s', e') \in A$. An \preceq -upward-closed set equals its upward closure. We denote by $\text{Min}(\preceq, A)$ the *minimal elements* of A , formally $\text{Min}(\preceq, A) := \{(s, e) \in A \mid \forall (s', e') \in A : (s', e') \preceq (s, e) \Rightarrow (s', e') = (s, e)\}$. A set $A \subseteq SE(c)$ is an *antichain* if all pairs $(s, e) \neq (s', e') \in A$ are \preceq -incomparable. An \preceq -upward-closed set is symbolically represented by its minimal elements, as the former is *uniquely* determined by the latter: if $A \subseteq SE(c)$ is an upward closed set, then $\text{Up}(\preceq, \text{Min}(\preceq, A)) = A$.

Description of the algorithm from [10] by operations on antichains Bouyer et al. [6] present an alternative description to the fixed point solution in terms of sufficient *infimum credits*. Since every

element U of the Cpre_c application from [10, Eqn.1] is \preceq -upward-closed, it can be symbolically represented by the antichain $\text{Min}(\preceq, U)$ [10]. For a bound $c \in \mathbb{N}$ the bounded operator $\text{CpreMin}_c : \mathcal{A}(SE(c)) \rightarrow \mathcal{A}(SE(c))$, where $\mathcal{A}(SE(c))$ is the set of antichains of $SE(c)$, is defined by:

$$\varepsilon_{\min}(\Lambda) = \{(s_0, e_0) \in SE(c) \mid s_0 \in \text{States}_0 \wedge e_0 = \min_{(s_0, s) \in E \text{ s.t. } \exists e_1(s, e_1) \in \Lambda} [\max(0, e_1 - w(s_0, s))]\} \quad (1)$$

$$\eta_{\min}(\Lambda) = \{(s_1, e_1) \in SE(c) \mid s_1 \in \text{States}_1 \wedge e_1 = \max_{(s_1, s) \in E \text{ s.t. } \exists e_0(s, e_0) \in \Lambda} [\max(0, e_0 - w(s_1, s))]\} \quad (2)$$

$$\text{CpreMin}_c(\Lambda) = \text{Min}(\preceq, \varepsilon_{\min}(\Lambda) \cup \eta_{\min}(\Lambda)) = \varepsilon_{\min}(\Lambda) \cup \eta_{\min}(\Lambda) \quad (3)$$

CpreMin_c is used to compute the sets of pairs consisting of a state and the *minimal* initial energy that suffices for player-0 to win for that state. For every pair of antichains $A, B \in \mathcal{A}(SE(c))$, we define the partial order \sqsubseteq such that $A \sqsubseteq B$ iff $\forall (s_b, e_b) \in B \exists (s_a, e_a) \in A$ such that $(s_a, e_a) \preceq (s_b, e_b)$. Note that $A \sqsubseteq B$ iff $\text{Up}(\preceq, B) \subseteq \text{Up}(\preceq, A)$. CpreMin_c is a monotone operator over the complete lattice $(\mathcal{A}(SE(c)), \sqsubseteq, \emptyset, \{(s, 0) \mid s \in \text{States}\})^2$. Therefore, there exists a *least* fixed point of CpreMin_c in this lattice that can be calculated by iterated applications of CpreMin_c to the antichain $ZS = \{(s, 0) \mid s \in \text{States}\}$. ZS assigns every state 0 initial energy, which is the minimal initial energy that is sufficient for player-0 to win a 0 steps game. In general, if Λ contains states and minimal energy levels for player-0 to win in k steps, then $\text{CpreMin}_c(\Lambda)$ contains those required for $k + 1$ steps.

3 Example Specification: Elevator

We present an example specification of a controller for an elevator servicing multiple floors. The environment inputs are pending requests to floors and the current floor of the cabin. The controller outputs are commands for moving up, stopping, or moving down. Quantitative properties of the specification are expressed as weights on transitions. Negative weights whenever the elevator is not on the requested floor and positive weights for reaching a requested floor.

Reactive specification The elevator specification is shown in List. 1. The environment controls the variable `pending` which signals a request of the elevator cabin to a destination floor given by the variable `dest_floor`. The environment also maintains variables to keep track of the source floor `src_floor` (location of cabin when request arrived) and the current floor `current_floor`. The system controls the variable `move` with the possible moves of the cabin (l. 7).

The specification in List. 1 defines two abbreviations `TOP` for the top floor and `THERE` for the condition that the current floor is the requested destination floor. The remainder of the specification defines from l. 13 to l. 26 safety constraints for the environment, i.e., assumptions, and from l. 28 to l. 31 safety constraints for the system, i.e., guarantees. Some important assumptions regard the values of variables storing the source floor: the value of source floor is the current floor when a request is issued (l. 14) and source floor and destination floor do not change while requests are pending (l. 16). Requests are disabled when the destination floor is reached (l. 18). The assumptions in l. 21 to l. 26 ensure that the environment sets the current floor according to the `move` commands of the system.

The safety constraints for the system ensure that the cabin does not move up on the top floor (l. 29) and that it does not move down on the bottom floor (l. 31).

²For every $M \subseteq \mathcal{A}(SE(c))$, $\inf M := \text{Min}(\preceq, \bigcup M)$, $\sup M := \text{Min}(\preceq, \bigcap_{m \in M} \text{Up}(\preceq, m))$. For details and proofs see [11].

```

1  VARENV
2  pending : boolean;
3  src_floor : 0..4;
4  dest_floor : 0..4;
5  current_floor : 0..4;
6  VAR
7  move : {UP, DOWN, STOP};
8
9  DEFINE
10 TOP := 4;
11 THERE := current_floor = dest_floor;
12
13 ASSUMPTION -- set src_floor
14   G (!pending & next(pending) -> next(src_floor) = next(current_floor));
15 ASSUMPTION -- remember src_floor and dest_floor
16   G (pending -> (next(src_floor) = src_floor & next(dest_floor) = dest_floor));
17 ASSUMPTION -- keep requests pending
18   G (pending & !THERE -> next(pending));
19 ASSUMPTION -- disable requests
20   G (THERE & pending -> next(!pending));
21 ASSUMPTION -- do go up
22   G (move=UP & current_floor < TOP -> next(current_floor) = current_floor + 1);
23 ASSUMPTION -- do stop
24   G (move=STOP -> next(current_floor) = current_floor);
25 ASSUMPTION -- do go down
26   G (move=DOWN & current_floor > 0 -> next(current_floor) = current_floor - 1);
27
28 GUARANTEE -- don't go up on top floor
29   G (current_floor = TOP -> move!=UP);
30 GUARANTEE -- don't go down on bottom floor
31   G (current_floor = 0 -> move!=DOWN);

```

Listing 1: A specification for an elevator controller consisting of environment variables VARENV, system variables VAR, definitions DEFINE, assumptions ASSUMPTION, and guarantees GUARANTEE

Symbolic elevator game graph The assumptions and guarantees in List. 1 describe the game graph for a reactive game in terms of initial states of the environment θ^e and system θ^s and the transitions of the environment ρ^e and system ρ^s . The conjunction of all assumptions without the temporal operator **G** defines θ^e while the conjunction of all guarantees without the temporal operator **G** defines θ^s . In the example of List. 1 we have no such assumptions or guarantees and thus $\theta^e \equiv \text{TRUE} \equiv \theta^s$. Similarly, the conjunction of all assumptions with the temporal operator **G** defines ρ^e while the conjunction of all guarantees with the temporal operator **G** defines ρ^s . The two guarantees in List. 1, ll. 28-31 express that ρ^s evaluates to FALSE for all states with `current_floor=TOP & move=UP` or `current_floor=0 & move=DOWN`, i.e., these states are deadlocks for the system.

A state in the game graph is an assignment to all variables, e.g. $s_1 = (\text{pending}=\text{true}, \text{src_floor}=1, \text{dest_floor}=4, \text{current_floor}=1, \text{move}=\text{DOWN})$. From this state the assignments to environment variables in successor states are determined by the transition relation ρ^e and the assignment to system variable `move` is restricted by the transition relation ρ^s to either `STOP` or `UP` leading to two possible successor states of s_1 . The number of reachable states in the game graph is 750. For 10 floors the number of reachable states is 6,000 and for 50 floors the number of reachable states is 750,000.

```

1 WEIGHT: 1
2   pending & next (!pending) & abs(src_floor - dest_floor) = 1;
3 WEIGHT: 2
4   pending & next (!pending) & abs(src_floor - dest_floor) = 2;
5 WEIGHT: 3
6   pending & next (!pending) & abs(src_floor - dest_floor) = 3;
7 WEIGHT: 4
8   pending & next (!pending) & abs(src_floor - dest_floor) = 4;
9 WEIGHT: -1
10  pending & !THERE;

```

Listing 2: A weight definition for transitions of the elevator with reward for reaching a floor depending on the distance traveled (between 0 and 4) and punishment -1 for not being at the requested floor. Five entries define six weights -1, 0, 1, 2, 3, and 4

```

1 WEIGHT: 1
2   pending & THERE;
3 WEIGHT: -1
4   pending & !THERE;

```

Listing 3: Weight definition for transitions of the elevator with reward 1 for reaching a requested floor and punishment -1 for not being at the requested floor. Two entries define three weights -1, 0, 1

Weight definitions List. 2 shows a definition of weights for transitions of the elevator. Every weights definition entry starts with keyword **WEIGHT** and the value of the weight followed by an LTL formula characterizing a set of transitions. As an example, the first entry in List. 2 defines weight 1 for transitions from states with a pending request and absolute difference **abs**(src_floor-dest_floor) = 1 to states with the request disabled. As another example, the last entry in List. 2 defines weight -1 for all transitions from states where a request is pending and the cabin is not at the destination floor. Intuitively, the weight definition of List. 2 expresses positive weights per distance traveled.

We also present an alternative weight definition in List. 3 consisting of two entries only. The first entry defines weight 1 for all transitions leaving states where a request is pending and the cabin is at the destination floor (l. 2). The second entry defines weight -1 for transitions from states with pending request where the cabin is not at the requested floor.

If a transition between states satisfies multiple weight definitions the values of all weights for the transition are added, e.g., a transition that satisfies both the formula in line 8 for weight 4 and the formula in line 10 for weight -1 in List. 2 has weight 3. In case a transition does not satisfy any of the weight formulas it is assigned the default weight 0, e.g., all transitions from states where no requests are pending in the weight definition of List. 3.

The addition of weights for overlapping sets of transitions and the completion with 0 is a pre-processing step. This means that the weights definition in List. 2 does not define 5 weights (number of **WEIGHT** entries) but 6 weights (including weight 0 for transitions defined both in line 2 and line 10). The weights definition in List. 3 defines 3 weights of values -1, 0, and 1. When referring to the number of weights, e.g., to describe the complexity of the algorithm the number of weights is the resulting number of non-overlapping weights and not the number of entries in our declarative weights specification.

Elevator energy game and initial energy The safety constraints of the system and environment in List. 1 together with the weight definition in List. 2 or List. 3 define an energy game. Intuitively, in an energy game the system starts with a finite amount of energy and has to make sure that in an infinite

play the accumulated energy does never drop below 0. The accumulated energy is defined based on the weights of transitions: for every combined step of the environment and the system (input and output) the weight definition defines an update of the energy level by adding the weight value.

For the first weights definition shown in List. 2 the system can win the energy game with a finite initial energy. An obvious strategy is to always immediately move to a floor once it is requested by the environment. The accumulated negative weights -1 for pending requests will then be compensated by the weight for reaching the destination floor. Interestingly, the elevator can do even better in terms of minimal energy levels. The highest minimal required initial energy from any state is 7. One such worst-case state is (`pending=true`, `src_floor=4`, `dest_floor=4`, `current_floor=1`, `move=DOWN`). In this state the cabin is on floor 1 but travels to floor 0 while the environment issued a request to the top floor.³ The top floor can be reached within 5 steps (accumulated energy level $7-5=2$). The reward for reaching the floor is 0 because `src_floor = dest_floor` (note that this reward is obtained on outgoing transitions). The cabin immediately travels to floor 3 where no request is pending (see assumption in List. 1, l. 20). It then arrives at floor 2 still with energy level 2 (because no request was pending on floor 3) and can now reach all floors within two steps and thus maintaining at least 0 energy.

For the alternative weights definition shown in List. 3 the system cannot win for any finite initial energy. From the above example strategy it is clear that with weight 2 instead of 1 in List. 3, l. 1 the system could win the energy game⁴.

4 Our Symbolic Algorithm

We start with a description of our *reactive* EG model in terms of the general model of EG, both presented in Sect. 2, where player-0 (maximizer) is the system and player-1 (minimizer) is the environment. Such a description shows that our model is an instance of the general model despite their differences. We then proceed with the presentation of a generic version of our symbolic algorithm for reactive EG.

4.1 Model of Reactive Energy Games

Formally, we show a reduction that takes as input a symbolic game graph $G^{sym} = \langle \theta^e, \theta^s, \rho^e, \rho^s, w \rangle$ for reactive EG as defined in Sect. 2, and outputs a *bipartite* game graph $\Gamma^R = \langle G^R = \langle V^R, E^R, w^R \rangle, V_0^R, V_1^R \rangle$ for EG. We start by defining two sub graphs of G^R , each consists of a simple cycle C such that if any of its vertices is reached during a play then the players are trapped in C indefinitely. (1) C^{win} : a (+1) weight cycle formed by the vertices $v_0^{win} \in V_0^R$, $v_1^{win} \in V_1^R$, and the edges $e_0^{win} = (v_0^{win}, v_1^{win}) \in E^R$, $e_1^{win} = (v_1^{win}, v_0^{win}) \in E^R$ such that $w^R(e_0^{win}) = 1$, $w^R(e_1^{win}) = 0$. (2) C^{loss} : a (-1) weight cycle formed by the vertices $v_0^{loss} \in V_0^R$, $v_1^{loss} \in V_1^R$, and the edges $e_0^{loss} = (v_0^{loss}, v_1^{loss}) \in E^R$, $e_1^{loss} = (v_1^{loss}, v_0^{loss}) \in E^R$ such that $w^R(e_0^{loss}) = (-1)$, $w^R(e_1^{loss}) = 0$.

Γ^R is constructed by the following two phases: (1) *Takes $\langle \theta^e, \theta^s \rangle$ as inputs*: we add an *initial* vertex $v_\emptyset \in V_1^R$ that corresponds to \mathcal{V}_\emptyset , i.e., all variables have no values assigned. In case there are no valid initial states for the environment, i.e., $\theta^e = \emptyset$, thus all plays on G^{sym} are winning for the system, we add $e = (v_\emptyset, v_0^{win}) \in E^R$ with $w^R(e) = 0$ leading to C^{win} , and output Γ^R . Otherwise, for every initial state $s^e \in \theta^e$ we add a vertex $v_{s^e} \in V_0^R$ and an edge $e = (v_\emptyset, v_{s^e}) \in E^R$ such that $w^R(e) = 0$, whereas for every $s \in \theta^e \cap \theta^s$ where $s = s_e \cup s_s$ we add a vertex $v_s \in V_1^R$ and an edge $e = (v_{s^e}, v_s) \in E^R$ with $w^R(e) = 0$.

³Note that this initial state is not required for winning because the system could choose `move=UP` but it represents an interesting worst-case energy level.

⁴This non-trivial strategy for winning with at most weight 2 for 5 floors was indeed found by our implementation.

For every $s^e \in \theta^e$ such that for all $s^s \in \mathcal{V}_{sys}$, $s^e \cup s^s \notin \theta^s$, i.e., an initial state which is a deadlock for the system, we add $e = (v_{s^e}, v_1^{loss}) \in E^R$ with $w^R(e) = (-1)$ leading to C^{loss} . (2) *Takes $\langle \rho^e, \rho^s, w \rangle$ as inputs:* for every valid environment transition $t_{s_1, s_2^e} \in \rho^e$, we add $v_{s_1} \in V_1^R$, $v_{t_{s_1, s_2^e}} \in V_0^R$ and $e = (v_{s_1}, v_{t_{s_1, s_2^e}}) \in E^R$ with $w^R(e) = 0$. For every $t_{s_1, s_2^e} \in \rho^e$ such that for all system choices s_2^s , $t_{s_1, s_2^e, s_2^s} \notin \rho^s$, i.e., it is a deadlock for the system, we add $e = (v_{t_{s_1, s_2^e}}, v_1^{loss}) \in E^R$ with $w^R(e) = (-1)$ leading to C^{loss} . For every $t_{s_1, s_2^e} \in \rho^e$ and system choice s_2^s such that $t_{s_1, s_2^e, s_2^s} \in \rho^s$, which results in a valid transition $t_{s_1, s_2} \in \rho^e \cap \rho^s$ for both players, we add $v_{s_2} \in V_1^R$ and $e = (v_{t_{s_1, s_2^e}}, v_{s_2}) \in E^R$ with $w^R(e) = w(s_1, s_2)$. For every state s_1 with $v_{s_1} \in V_1^R$ such that for all environment choices s_2^e , $t_{s_1, s_2^e} \notin \rho^e$, i.e., a deadlock state for the environment, we add $e = (v_{s_1}, v_0^{win}) \in E^R$ with $w^R(e) = 0$ leading to C^{win} . Γ^R has the following properties:

Lemma 1. *Given an initial credit $c \in \mathbb{N}$, $\theta^e = \emptyset$ if, and only if for all EG plays p on Γ^R , $p = v_0(v_0^{win} v_1^{win})^\omega$, and for all $k \geq 1$, $EL(p, k) \geq 0$.*

Lemma 2. *$s^e \in \theta^e$ is an initial deadlock state for the system if, and only if $v_{s^e} \in V_0^R$ has one outgoing (-1) weighted edge to $v_1^{loss} \in V_1^R$ in C^{loss} and there is no initial energy value $c \in \mathbb{N}$ sufficient for player-0 to win from $v_0 \in V_1^R$.*

Lemma 3. *$s \in \theta^e \cap \theta^s$ is an initial deadlock state for the environment in G^{sym} if, and only if $v_s \in V_1^R$ has one outgoing 0 weighted edge to $v_0^{win} \in V_0^R$ in C^{win} and every initial energy value $c \in \mathbb{N}$ suffices for player-0 to win from v_s .*

Lemma 4. *A valid transition from s_1 for environment choice s_2^e , $t_{s_1, s_2^e} \in \rho^e$, is a deadlock for the system if, and only if $v_{t_{s_1, s_2^e}} \in V_0^R$ has one outgoing (-1) weighted edge to $v_1^{loss} \in V_1^R$ in C^{loss} and there is no initial energy value $c \in \mathbb{N}$ sufficient for player-0 to win from $v_{s_1} \in V_1^R$.*

Lemma 5. *s_2 is a deadlock state for the environment in G^{sym} with a predecessor state s_1 such that $t_{s_1, s_2} \in \rho^e \cap \rho^s$ if, and only if $v_{s_2} \in V_1^R$ has one outgoing 0 weighted edge to $v_0^{win} \in V_0^R$ in C^{win} and every initial energy value $c \in \mathbb{N}$ suffices for player-0 to win from v_{s_2} .*

Lemma 6. *A play on G^{sym} starts at $s \in \theta^e \cap \theta^s$ if, and only if a play on Γ^R starts with the traversal of $e_1 = (v_0, v_{s^e}) \in E^R$ by player-1 and $e_0 = (v_{s^e}, v_s) \in E^R$ by player-0, with $w^R(e_1) = w^R(e_0) = 0$.*

Lemma 7. *A valid $W \in \mathbb{Z}$ weighted transition for both players from s_1 to s_2 , i.e., $t_{s_1, s_2} \in \rho^e \cap \rho^s$, is taken at step $j \in \mathbb{N}$ of a play p^{sym} on G^{sym} if, and only if at step $2j + 2$ of a play p on Γ^R player-1 traverses $e_1 = (v_{s_1}, v_{t_{s_1, s_2^e}}) \in E^R$ and player-0 traverses $e_0 = (v_{t_{s_1, s_2^e}}, v_{s_2}) \in E^R$ such that $w^R(e_1) = 0$ and $w^R(e_0) = W$.*

Theorem 1. ⁵ *Given an initial credit $c \in \mathbb{N}$, for all $j \geq 1$, $e \in \mathbb{Z}$, there exists an infinite reactive EG play $p^{sym} = s_0 s_1 \dots s_{j-1} s_j \dots$ on G^{sym} with $EL(p^{sym}, j) = e$ if, and only if there exists an EG play $p = v_0 v_{s_0^e} v_{s_0} v_{t_{s_0, s_1^e}} v_{s_1} \dots v_{s_{j-1}} v_{t_{s_{j-1}, s_j^e}} v_{s_j} \dots = v_0 v_1 v_2 v_3 v_4 \dots v_{2j} v_{2j+1} v_{2j+2} \dots$ on Γ^R with $EL(p, 2 + 2j) = e$.*

Theorem 2. ⁵ *Given an initial credit $c \in \mathbb{N}$, for all $e \in \mathbb{N}$, $n \in \mathbb{N}$, $1 \leq j \leq n$, there exists a finite winning reactive EG play p^{sym} on G^{sym} that ends with a deadlock for the environment, $p^{sym} = s_0 s_1 \dots s_{n-1} s_n$, with $EL(p^{sym}, j) = e$ if, and only if there exists an EG play $p = v_0 v_{s_0^e} v_{s_0} v_{t_{s_0, s_1^e}} v_{s_1} \dots v_{s_{n-1}} v_{t_{s_{n-1}, s_n^e}} v_{s_n} (v_0^{win} v_1^{win})^\omega = v_0 v_1 v_2 v_3 v_4 \dots v_{2n} v_{2n+1} v_{2n+2} (v_0^{win} v_1^{win})^\omega$ on Γ^R with $EL(p, 2 + 2j) = e$ and for all $k \geq 1$, $EL(p, k) \geq 0$.*

Theorem 3. ⁵ *Given an initial credit $c \in \mathbb{N}$, there exists a finite losing reactive EG play p^{sym} on G^{sym} that ends with $s^e \in \theta^e$ which is a deadlock for the system, $p^{sym} = s^e$, if, and only if there exists an EG play $p = v_0 v_{s^e} (v_1^{loss} v_0^{loss})^\omega$ on Γ^R , and there exists $k \geq 1$ such that $EL(p, k) < 0$.*

⁵We omit the proof from this submission.

Theorem 4. ⁵ Given an initial credit $c \in \mathbb{N}$, for all $e \in \mathbb{Z}$, $n \in \mathbb{N}$, $1 \leq j < n$, there exists a finite losing reactive EG play p^{sym} on G^{sym} that ends with a deadlock for the system, $p^{sym} = s_0 s_1 \dots s_{n-1} s_n^e$, with $EL(p^{sym}, j) = e$ if, and only if there exists an EG play $p = v_0 v_{s_0^e} v_{s_0} v_{t_{s_0, s_1^e}} v_{s_1} \dots v_{s_{n-1}} v_{t_{s_{n-1}, s_n^e}} (v_1^{loss} v_0^{loss})^\omega = v_0 v_1 v_2 v_3 v_4 \dots v_{2n} v_{2n+1} (v_1^{loss} v_0^{loss})^\omega$ with $EL(p, 2+2j) = e$, and there exists $k \geq 1$ such that $EL(p, k) < 0$.

From Lem. 1 - 5 we get that Γ^R has no deadlocks, and conclude from Thm. 1 - 4 that every EG play on Γ^R is infinite, therefore compliant with the general model of EG. Moreover, by Thm. 1 every play on Γ^R in which neither C^{loss} nor C^{win} are visited, determines a reactive play on G^{sym} with the *same* energy levels, and vice versa. From Thm. 2 and Lem. 1, 3, 5 we get that every EG (winning) play on Γ^R that visits C^{win} determines a finite (winning) play on G^{sym} that ends with a deadlock for the environment. From Thm. 3, 4 and Lem. 2, 4 we get that every EG play on Γ^R that visits C^{loss} is losing for the system, and it determines a finite losing game on G^{sym} that ends with a deadlock for the system. Thm. 1 - 4 focus on the energy levels of player-1's vertices in Γ^R , which are the ones of interest for our reactive EG, and imply that an initial energy level $c \in \mathbb{N}$ suffices for player-0 to win from $v_s \in V_1^R$ in Γ^R iff c suffices for player-0 to win from a state s which is reachable from any of the valid initial states of G^{sym} .

4.2 Generic Version of Our Algorithm

Alg. 1 presents a generic version of our symbolic algorithm for reactive EG, which takes as input an energy bound $\maxEng \in \mathbb{N}$. It performs within the while loop in line 3 a least fixed point calculation of $CpreMin_c$ operator from Eqn. (3) by its iterated applications to the antichain $\{(s, 0) \mid s \in \text{States}\}$, initially assigned to \minEngPred in line 2. However, the calculation is *optimized* for our model by means of a different formulation of Eqn. (3). We present $CpreMin_c^{OPT}$ for a reactive EG graph Γ^R as constructed by the reduction of Sect. 4.1, where $SE_1^R(c) := V_1^R \times \{n \in \mathbb{N} \mid n \leq c\}$ for $c \in \mathbb{N}$:

$$CpreMin_c^{OPT}(\Lambda) = \{(v_1, e_1) \in SE_1^R(c) \mid e_1 = \max_{(v_1, v_0) \in ER} \left(\min_{(v_0, v'_1) \in ER_{s.t.} \exists e'_1 (v'_1, e'_1) \in \Lambda} \left(\max(0, e'_1 - w^R((v_0, v'_1))) \right) \right)\} \quad (4)$$

The $CpreMin_c^{OPT}$ operator from Eqn. (4), applies Eqn. (1) followed by Eqn. (2), i.e., η_{min} is applied to the minimal energy values that have just been calculated by ϵ_{min} in the current iteration. This optimization utilizes the reactive property of the model, i.e., its turn alternation that induces a bipartite game graph, by which the initial energy values of player- i 's vertices calculated in iteration k only depend on the values of player- j 's vertices calculated in iteration $k-1$, $i \neq j$. Moreover, for optimizing Eqn. (2), it utilizes the property of 0 weight for all outgoing edges from all $v \in V_1^R$. We denote by A_i^{OPT} and A_i the i 'th element of the chain results from the least fixed point computation on Γ^R of Eqn. (4) and Eqn. (3), respectively. Lem. 8 formally states that these two operators are equivalent when applied to Γ^R , while the number of iterations (i.e., the chain's length) until a fixed point is reached is smaller by a factor of 2 for $CpreMin_c^{OPT}$.

Lemma 8. Given an initial credit $c \in \mathbb{N}$, for all $i \geq 0$, $A_{2i} \cap SE_1^R(c) = A_i^{OPT}$.

We present in line 5 of Alg. 1 the symbolic formulation of Eqn. (4), denoted by $CpreMin_c^{symOPT}$, applied to G^{sym} . It invokes the function f of Eqn. 5 which handles deadlock cases where w is undefined. We denote by A_i^{symOPT} the i 'th element of the chain resulting from its iterative application by Alg. 1.

$$f(s, s^e, s^s, e', \maxEng) = \begin{cases} 0 & \text{if } t_{s, s^e} \notin \rho^e \\ \maxEng + 1 & \text{if } t_{s, s^e, s^s} \notin \rho^s \\ \max[0, e' - w(s, s^e \cup s^s)] & \text{otherwise} \end{cases} \quad (5)$$

Algorithm 1 Generic symbolic fixed point algorithm for reactive energy games played on a symbolic game graph G^{sym} with initial energy bound $\maxEng \in \mathbb{N}$ using function f as defined in Eqn. (5)

```

1: define minEngStates, minEngPred as (States  $\times$   $\mathbb{N}$ )
2: minEngPred =  $\{(s, 0) \mid s \in \text{States}\}$ 
3: while minEngStates  $\neq$  minEngPred do
4:   minEngStates = minEngPred
5:   minEngPred =  $\{(s, e) \in \text{States} \times \{0, 1, \dots, \maxEng\} \mid$ 

$$e = \max_{s^e \in \mathcal{V}_{env}} \left( \min_{s^s \in \mathcal{V}_{sys} \text{ s.t. } \exists e'. (s^e \cup s^s, e') \in \text{minEngStates}} (f(s, s^e, s^s, e', \maxEng)) \right)$$

6: end while
7: return minEngStates

```

Theorem 5 (Correctness and Completeness of Alg. 1). *Alg. 1 computes the minimal energy value for each state within the bound \maxEng .*

Proof. By Lem. 8, if A_i^{OPT} contains states and minimal energy levels for player-0 to win in $2i$ steps of a play on Γ^R , then A_{i+1}^{OPT} contains those required for $2i + 2$ steps. We infer from Lem. 7 that it is equivalent to an increment of a single step in the respective play on G^{sym} . Therefore, from Thm. 1 - 4 we get that for every $i \geq 0$, $e \in \mathbb{N}$, and for every state s of G^{sym} and its respective vertex $v_s \in V_1^R \setminus \{v_\emptyset, v_1^{win}, v_1^{loss}\}$: $(s, e) \in A_i^{symOPT}$ iff $(v_s, e) \in A_i^{OPT}$. It shows that line 5 implements Eqn. 3 and thus the algorithm by Chatterjee et al. [10] for the special case of reactive EG. \square

Checking Realizability Alg. 1 computes a set of winning states and their required minimal initial energy. To check whether the system can win the energy game we still have to check whether for every initial choice of the environment described by θ^e the system has an initial choice satisfying θ^s to select a winning state. For winning states $win = \{s \mid (s, e) \in \text{minEngStates}\}$ this check is $\forall s^e \exists s^s : s^e \in \theta^e \Rightarrow s^e \cup s^s \in \theta^s \cap win$. The check has a direct implementation in BDDs and ADDs.

Running Time Complexity The running time complexity of Alg. 1 in symbolic steps is in $O(N \cdot \maxEng)$. The number of iterations of the while loop is bounded by the number of states N times the bound \maxEng , i.e., the number of possible unique configurations of the monotonic minEngStates . A worst case example that does indeed require $N \cdot \maxEng + 2$ fixed point iterations — it changes the value of a single state by value 1 in every iteration except in the first and last iterations — is a cycle of uneven length N (number of states) with weight 1 from uneven to even states and weight -1 otherwise.⁶

5 BDD Algorithm

We present a BDD-based implementation of Alg. 1 to compute the minimal initial energy for every state. Alg. 2 takes as input a bound $\maxEng \in \mathbb{N}$, a definition of weights $\text{weights} \subseteq (\mathbb{Z} \times \text{Set of Transitions})$ which is an implementation of the weight function w^7 as defined in Sect. 2, and returns a relation $\text{minEngStates} \subseteq (\mathbb{N} \times \text{Set of States})$ of pairs of initial required energy and winning states. The result

⁶The worst case behavior here is due to unrealizability. The same game on a cycle of even length N requires two iterations.

⁷Every weight appears once with all transitions of that weight.

Algorithm 2 BDD algorithm for minimal required energy per set of states for environment transitions ρ^e , system transitions ρ^s , $\text{weights} \subseteq (\mathbb{Z} \times \text{Set of Transitions})$, and energy bound $\text{maxEng} \in \mathbb{N}$.

```

1: define minEngStates, minEngPred as  $(\mathbb{N} \times \text{Set of States})$ 
2: add  $(0, \text{TRUE})$  to minEngPred
3: while minEngStates  $\neq$  minEngPred do
4:   minEngStates = minEngPred
5:   empty minEngPred
6:   remaining =  $\{s \in S \mid (e, S) \in \text{minEngStates}\}$ 
7:   for increasing best  $\in \{0\} \cup \{0 \leq e - w \leq \text{maxEng} \mid (e, S) \in \text{minEngStates} \wedge (w, T) \in \text{weights}\}$  do
8:     define bestT as Transitions
9:     bestT =  $\emptyset$ 
10:    for  $(v, T) \in \text{weights}$  do
11:       $S = \{s \in S_{e_v} \mid (e_v, S_{e_v}) \in \text{minEngStates} \wedge e_v - v \leq \text{best}\}$ 
12:      add T transition to S to bestT
13:    end for
14:    B = (forceEnvTo bestT transitions)  $\cap$  remaining
15:    add (best, B) to minEngPred
16:    remove B from remaining
17:  end for
18: end while
19: return minEngStates

```

is empty if no state exists that is winning for initial energy up to maxEng . States and transitions are implemented as BDDs over the variables $\text{env} \cup \text{sys}$ and $\text{env} \cup \text{sys} \cup \text{env}' \cup \text{sys}'$, respectively.

Alg. 2 declares two relations of required energy and states and initializes minEngPred with energy 0 for all possible states (TRUE). The main while loop in line 3 implements the fixpoint computation of Alg. 1. The algorithm stores the result of the last computation in variable minEngStates and empties the current relation minEngPred in lines 4 and 5. The variable remaining is assigned the union of states that can maintain at least 0 energy in k steps, i.e., the states from minEngStates (l. 6).

In the for-loop from line 7 to line 17 the algorithm determines for states in remaining the value best for $k+1$ steps. It computes the minimal value best such that the system can force the environment to use only transitions where the required energy of the successor for k steps minus the weight of the transition is at most best . The value best is a combination of required energy values from minEngStates and weights from weights (see line 7). The value 0 is always included and contributes deadlock states of the environment.

The inner for-loop from line 10 to line 13 collects all combinations of transitions to target states bestT that require best energy value or less for predecessor states. The algorithm iterates over all weight and transition pairs (v, T) and for every weight selects in line 11 successor states S with accumulated energy e_v such that reaching a selected successor requires $e_v - v \leq \text{best}$ energy for $k+1$ steps. In line 12 the combination of T transitions with S target states is added to bestT . This addition is implemented in BDD operations as $\text{bestT} = \text{bestT} \vee (T \wedge \text{prime}(S))$.

Finally, the algorithm computes the set B of remaining predecessor states from which the system can restrict the environment to take only bestT transitions in line 14. The method **forceEnvTo** bestT **transitions** is implemented in BDD operations as $(\rho^e \Rightarrow (\rho^s \wedge \text{bestT}))_{\exists \text{sys}' } \forall \text{env}'$. Thus, all states in B require at most best energy for $k+1$ steps. This is also the minimal value because the algorithm

searched by increasing value of `best`. The states `B` are removed from the `remaining` states.

Running Time Complexity The running time complexity of Alg. 2 in symbolic steps is in $O(N \cdot \text{maxEng}^2 \cdot |\text{weights}|)$. The iterations of the while loop in line 3 are in $O(N \cdot \text{maxEng})$ as discussed in Sect. 4. The iterations of every execution of the for loop in line 7 are bounded by `maxEng` because $0 \leq \text{best} \leq \text{maxEng}$. Every execution of the innermost for loop executes the loop body $|\text{weights}|$ times (once for every distinct weight). When a fixpoint is reached, i.e., `minEngStates` = `minEngPred`, the algorithm terminates with the last computed result of `minEngStates`.

For the example of the elevator specification in List. 1, the weight definition of `weights` from -1 to 4 shown in List. 2, and `maxEng` = 100 we have $N = 750$ and $|\text{weights}| = 6$. The algorithm reaches a fixed point already after 6 iterations of the outer most while loop in less than a second.

Correctness and completeness Alg. 2 implements Alg. 1. The update operation in Alg. 1, l. 5 is implemented by Alg. 2, ll. 6-16. The minimum is implemented by starting with smallest `best` and the maximum is implemented by increasing `best` until the environment cannot force higher values for `best`. Therefore, by Thm. 5 it computes the minimal energy value for each state within the bound.

6 ADD algorithm

Alg. 3 shows our ADD-based implementation of Alg. 1. It takes as input a bound `maxEng` $\in \mathbb{N}$, a function `weights` : `Transitions` $\rightarrow \mathbb{Z}$ which is an ADD implementation of the weight function w as defined in Sect. 2, assigning weights to valid transitions for both players, and ρ^e and ρ^s as defined in Sect. 2. It returns a function `minEngStates` : `States` $\rightarrow \mathbb{N} \cup \{+\infty\}$ that assigns every state the minimal required initial energy, or a $+\infty$ value if it is not winning for initial energy up to `maxEng`. States and transitions are implemented as ADDs over the variables `env` \cup `sys` and `env` \cup `sys` \cup `env'` \cup `sys'`, respectively. We denote the value of an ADD a : `Transitions` \rightarrow Values for a transition from s_1 to $s_2 \in$ `States` by $a(s_1, s_2)$.

Alg. 3 declares two functions, `minEngStates` and `minEngPred`, assigning every state the minimal required initial energy. Line 6 initializes `minEngPred` as the constant 0 function for all states, which is the minimal energy level sufficient for a 0 steps play. Alg. 3 also constructs an ADD representing the weighted game graph, `arena`, which models invalid transitions, i.e., deadlocks, by extending `weights` such that every invalid environment transition, i.e., $t_{s,s^e} \notin \rho^e$, is assigned a $+\infty$ weight (system wins), and every invalid system transition, i.e., $t_{s,s^e,s^s} \notin \rho^s$, a $-\infty$ weight (system loses). The `arena` construction is implemented by two *if-then-else* operations for ADDs described in lines 4 and 5.

The while loop in line 7 performs the same computation as of Alg. 1, a least fixed point computation of the operator defined in Alg. 1, l. 5, but formulates it as a function rather than a relation due to its implementation by ADD operations. Each iteration assigns the resulted function of the previous computation for k steps to `minEngStates`, and computes an updated function `minEngPred` for $k+1$ steps by three ADD operations. The first operation in line 9 whose output is assigned to `accumulatedEng` implements the innermost arithmetic computation of Alg. 1, l. 5. It computes the energy levels of states and transitions to successor states in `minEngStates`⁸. The operator \ominus_{maxEng} is an implementation of Eqn. 5 compliant with the deadlocks representation in `arena`, defined as:

⁸Technically, to refer to `minEngStates` as successor states its ADD is primed, i.e., the `sys` and `env` variables of `minEngStates` are replaced by their primed counterparts.

Algorithm 3 ADD algorithm for minimal required energy per state for environment transitions ρ^e , system transitions ρ^s , $\text{weights} : \text{Transitions} \rightarrow \mathbb{Z}$, and energy bound $\text{maxEng} \in \mathbb{N}$.

```

1: define minEngStates, minEngPred as ( $\text{States} \rightarrow \mathbb{N} \cup \{+\infty\}$ )
2: define arena, deadlocks as ( $\text{Transitions} \rightarrow \mathbb{Z} \cup \{-\infty, +\infty\}$ )
3: define accumulatedEng as ( $\text{Transitions} \rightarrow \mathbb{N} \cup \{+\infty\}$ )
4: deadlocks( $s, s'$ ) = if  $\rho^e(s, s')$  then  $-\infty$  else  $+\infty$ 
5: arena( $s, s'$ ) = if  $(\rho^e \cap \rho^s)(s, s')$  then  $\text{weights}(s, s')$  else deadlocks( $s, s'$ )
6: minEngPred( $s$ ) = 0
7: while minEngStates  $\neq$  minEngPred do
8:   minEngStates = minEngPred
9:   accumulatedEng( $s, s'$ ) = minEngStates( $s'$ )  $\ominus_{\text{maxEng}}$  arena( $s, s'$ )
10:  minEngPred( $s$ ) =  $\max_{s^e \in \mathcal{V}_{\text{env}}} \min_{s^s \in \mathcal{V}_{\text{sys}}} \text{accumulatedEng}(s, s^e \cup s^s)$ 
11: end while
12: return minEngStates

```

$$f_1(s') \ominus_{\text{maxEng}} f_2(s, s') = \begin{cases} 0 & \text{if } f_2(s, s') = +\infty \\ +\infty & \text{if } f_1(s') = +\infty \text{ or } f_2(s, s') = -\infty \\ +\infty & \text{if } f_1(s') - f_2(s, s') > \text{maxEng} \\ \max[0, f_1(s') - f_2(s, s')] & \text{otherwise} \end{cases} \quad (6)$$

In iteration k , for each source and target states, Eqn. (6) results in the required accumulated energy level for k steps, that is the subtraction of the corresponding transition's weight from the target state's minimal required energy for $k - 1$ steps. If the transition's weight is $+\infty$, i.e., winning for the system, it results in a 0 value. Otherwise, unless it exceeds the maxEng bound, which results in a $+\infty$ value (i.e., this transition is losing for the given bound in k steps), a max operator (with 0 as its second argument) ensures the resulted accumulated energy is non-negative. Line 10 implements the min operator followed by the outermost max operator of Alg. 1, l. 5 by two ADD abstractions of `accumulatedEng` for successor system and environment choices⁹.

Running Time Complexity The running time complexity of Alg. 3 in symbolic steps is as the number of iterations of the while loop in line 7 which is in $O(N \cdot \text{maxEng})$, as discussed in Sect. 4. For the elevator specification in List. 1, weights from List. 2, and $\text{maxEng} = 100$ the algorithm reaches a fixed point after 6 iterations (same number of iterations as the BDD implementation in Alg. 2).

Correctness and Completeness Alg. 3 formulates Alg. 1 in terms of functions rather than relations. Therefore, by Thm. 5 it computes the minimal energy value for each state within the bound.

7 Implementation and Preliminary Evaluation

We have implemented Algorithms 2 and 3 in Java, based on an extended version of JTLV [20]. We have extended JTLV with support for ADDs as provided by CUDD. We use CUDD via JNI.

⁹Technically, the ADD minimum abstraction is done on variables sys' and the maximum abstraction on variables env' .

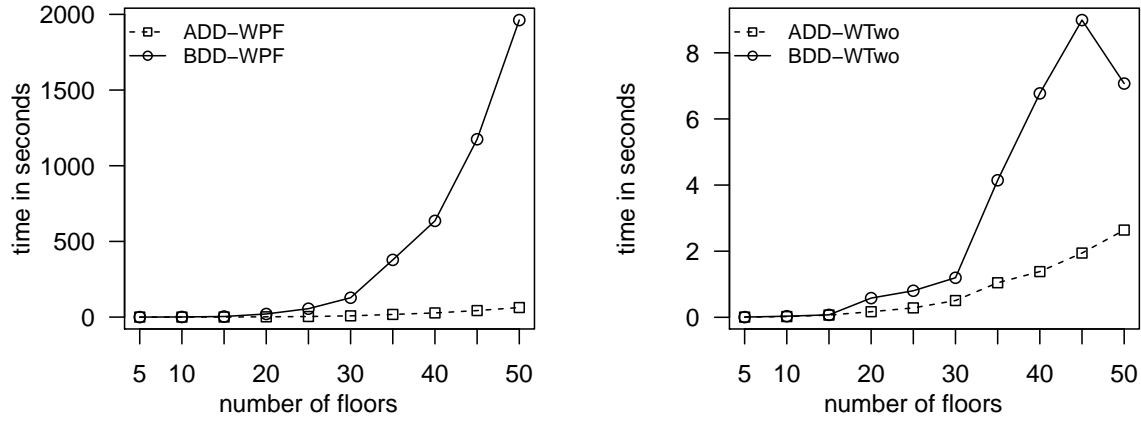


Figure 1: Running times of ADD and BDD algorithms on elevator specification from List. 1 with increasing number of floors from 5 to 50 and the energy bound $\text{maxEng} = 100$. **Left:** Weight definition of List. 2 with weights per floor (WPF) adapted to 5 to 50 floors. **Right:** Weight definition of List. 3 with two weight entries (WTwo) adapted to 5 to 50 floors.

We show a comparison of the two algorithms for computing the minimal required energy from all winning states, using the elevator example from Sect. 3. The questions our evaluation addresses are:

- How do the ADD and BDD algorithms scale for elevator specifications with increasing floors?
- Do they scale differently for the weight specification with many or few weights?
- What is the impact of the choice of the bound maxEng on the running time?

We run all experiments on an ordinary PC, Intel i7 CPU 3.4GHz, 16GB RAM with Windows 7 64-bit OS. Our implementation uses CUDD 2.5.0 compiled for 32Bit. The algorithms are implemented in a modified version of JTLV in Java 7 32Bit. Our implementations are not distributed and use only a single core of the CPU. We have run the algorithms on every specification 12 times and report average times if not stated otherwise. All times are wall-clock times as measured by Java.¹⁰

Increasing Number of Floors To measure how the BDD and ADD algorithms scale on the elevator specification when increasing the number of floors, we have created specifications similar to the one shown in List. 1 with increased numbers of floors in steps of 5 from 5 to 50 floors. The specification for n floors differs from List. 1 in ll. 3-5 in an updated range of the floor variables to $0 \dots n-1$ and in l. 10 updated to the new top floor $\text{TOP} := n-1$.

For our experiments we have used the weight definition in List. 2 with positive weight values of the distance traveled to each requested floor. The updated weight definition for n floors has n entries and defines $n+1$ weights from -1 to $n-1$. For all experiments we have chosen the bound maxEng of the maximal initial energy per state to be 100, independent of the number of floors. This bound makes the elevator specification realizable for all numbers of floors in the experiment.

The average running times for each specification are summarized in Fig. 1 (left). For each number of floors we present the data points for the ADD algorithm and the BDD algorithm, measured in seconds.

¹⁰The automatic variable reordering of CUDD made running times extremely unpredictable with time differences up to a factor of five when executing the same synthesis problem. Therefore, we have disabled any reordering of BDD and ADD variables. Both implementations use the order of variables as they appear in the specification. Without reordering the running times we measured appeared to be more stable.

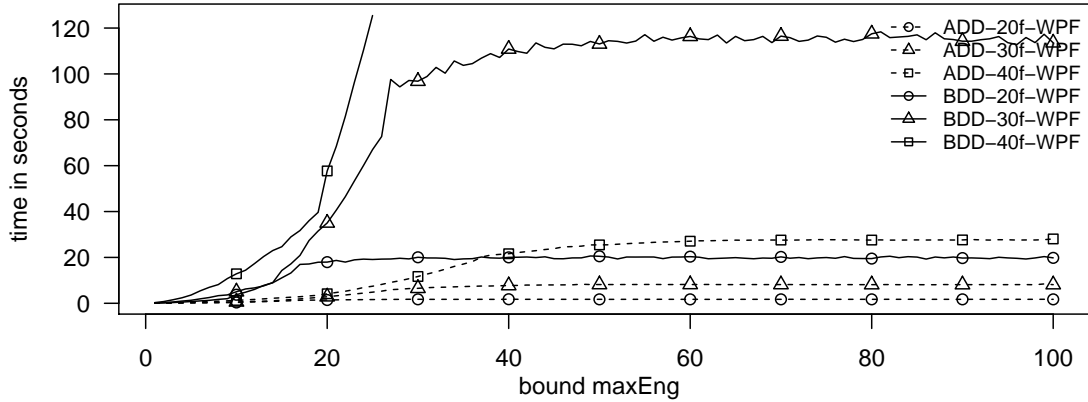


Figure 2: Running times of ADD and BDD algorithms on elevator specification from List. 1 for 20, 30, and 40 floors. Weight definition from List. 2 for bounds maxEng from 1 to 100 (specification with 20/30/40 floors becomes realizable for maxEng bound 36/56/76).

Up to 10 floors the running time is below a second. The running time for 50 floors of the ADD algorithm is around one minute while the BDD algorithm runs for 33 minutes.

From Fig. 1 (left) it is very clear that the ADD algorithm scales much better than the BDD algorithm for elevator specifications with increasing number of floors.

Different Weight Specifications To compare the running time of both algorithms for different weight specifications we have executed the same experiment as in Sect. 7 with the alternative weight definition shown in List. 3. For the elevator with n floors we have changed the single positive weight for reaching a requested floor in List. 3, l. 1 to n . In this experiment the number of floors ranges again from 5 to 50 in steps of 5 and the number of weights is always 3 (-1 , 0 , and n).

The average running times for each specification are summarized in Fig. 1 (right). Again, the ADD algorithm performed much better than the BDD algorithm. When comparing absolute running times shown in Fig. 1 (left) and (right) it is clear that a different weight specification for the same synthesis problem can make a significant difference in running times. For 50 floors the difference in running times between 3 weights defined in List. 3 and 51 weights defined in List. 2 is of factor 277 for the BDD algorithm and of factor 24 for the ADD algorithm.

Choice of Bound maxEng The two algorithms we have implemented are both bounded by the maximal initial energy maxEng . We are thus interested in the influence of the bound on the running times of the algorithms on specifications of the elevator. We run both the ADD and the BDD algorithm for increasing bounds maxEng .

Fig. 2 shows running times for the elevator specifications with 20/30/40 floors and the weights definition with positive weights for the distance traveled to a requested floor from List. 2. The x-axis shows the bound maxEng from weight 1 to 100 in steps of 1 (markers every 10 steps to distinguish 20/30/40 floors). The results are based on a single run for each specification, algorithm, and bound (600 runs) and thus may contain some noise, e.g., see the BDD algorithm for 30 floors with maxEng greater than 20. We have omitted the running times of the BDD algorithm above 120 seconds (running times go up to 640 seconds and remain above 600 seconds for increasing bound). The specifications for 20/30/40 floors are realizable for bounds of at least 36/56/76.

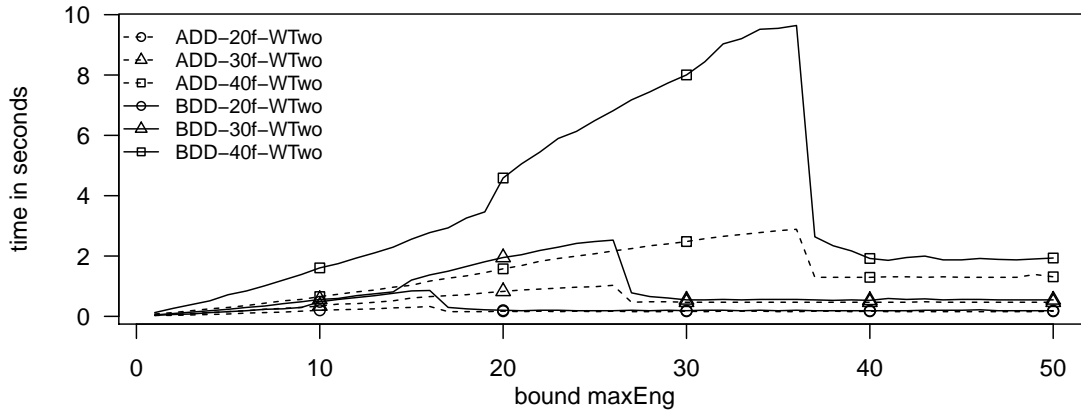


Figure 3: Running times of ADD and BDD algorithms on elevator specification from List. 1 for 20, 30, and 40 floors. Weight definition from List. 3 for bounds maxEng from 1 to 50 (specification with 20/30/40 floors becomes realizable for maxEng bound 19/29/39).

Fig. 3 shows running times for the elevator specifications with 20/30/40 floors and the weights definition with three weights from List. 3. The x-axis shows the bound maxEng from weight 1 to 50 in steps of 1 (markers every 10 steps to distinguish 20/30/40 floors). The results are based on a single run for each specification, algorithm, and bound (300 runs). The specifications for 20/30/40 floors are realizable for bounds of at least 19/29/39.

From both figures we observe that the ADD and BDD algorithms behave similarly. We see that the running times become stable a few runs before the bound for realizability is reached. For the weight definition of List. 2 with more weights the running times shown in Fig. 2 become stable for both algorithms much before the synthesis problem becomes realizable. Interestingly, for the weight definition of List. 3 and running times shown in Fig. 3, running times increase up to a factor of five before the problem becomes realizable and then drop again to become stable.

8 Discussion and Related Work

The evaluation in Sect. 7 shows that the ADD algorithm scales well for increasing the statespace of the synthesis problem. Both algorithms however show a strong increase in running times for weight definitions with more weights. The increase in running times is stronger for the BDD algorithm.

We have evaluated the two algorithms on specifications with very restricted game graphs. States with a pending request have a single environment successor and at most three system choices. These very sparse game graphs might not be suited to demonstrate the potential of symbolic computations. Experiments covering a larger set of specifications are required to better evaluate the potential of our symbolic implementations. Future analysis should also take memory consumption into account.

Energy games, as introduced by Bouyer et al. [6], were generalized to multi-dimensional energy games by Chatterjee et al. [9]. The energy game algorithm by Brim et al. [7], which we have extended in our work, has later been extended by Chatterjee et al. [10] to also solve multi-dimensional energy games. Bohy et al. [4] presented an algorithm for LTL synthesis via a k bounded reduction through universal k -co-Büchi automata. The algorithm is implemented in Acacia+ [3] and is symbolic in the multi-dimensional energy level and bound k . Since their input is LTL and weights are defined on atomic propositions and not on transitions, our implementations are not easily comparable. A multi-dimensional

extension might be possible for the algorithms we presented but it appears less natural for the ADD algorithm where terminals describe minimal energy levels for a single dimension.

Finally, another quantitative game in recent interest are Mean Payoff Games (MPG), introduced by Ehrenfeucht and Mycielski [13]. Bouyer et al. [6] showed that MPG and EG are log-space equivalent. Therefore, by applying a log-space reduction from MPG to EG, our two proposed symbolic algorithms for EG can also be used to solve MPG.

9 Conclusion

We have presented two algorithms for solving reactive energy games. Given an energy game and a maximal initial energy level both algorithms compute the minimal required energy per state for the system to maintain a positive energy level in an infinite play. The algorithms are optimized for reactive energy games in the sense that they update energy levels of system and environment states in one instead of two steps. To the best of our knowledge the algorithms are the first that are fully symbolic both in the energy levels and in the representation of the underlying safety game.

Our first algorithm uses BDDs while the second is implemented using ADDs. Both make specific use of symbolic manipulations for efficiently performing quantitative operations. We have compared the running times of the two implementations in a preliminary evaluation and found better scalability for the ADD algorithm for both extending the statespace and the number of distinct weight values. Our next step will be to evaluate the performance of both algorithms on a larger body of specifications, to implement and evaluate symbolic strategy construction, and to deal with the unrealizable case.

Our work shows that purely symbolic implementations for solving energy games are possible and might make quantitative synthesis more accessible and realistic for reactive systems engineers.

The work is part of a larger project on bridging the gap between the theory and algorithms of reactive synthesis on the one hand and software engineering practice on the other. As part of this project we are building engineer-friendly tools for reactive synthesis, see, e.g., [17, 18].

Acknowledgments We thank Yaron Velner for helpful introduction to and discussions about quantitative games. We thank Jean-Francois Raskin for encouraging discussions and implementation tips at the Marktoberdorf Summer School 2015. Part of this work was done while SM was on sabbatical as visiting scientist at MIT CSAIL. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 638049, SYNTech).

References

- [1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo & Fabio Somenzi (1997): *Algebraic Decision Diagrams and Their Applications*. *Formal Methods in System Design* 10(2/3), pp. 171–206, DOI: 10.1023/A:1008699807402.
- [2] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger & Barbara Jobstmann (2009): *Better Quality in Synthesis through Quantitative Objectives*. In Bouajjani & Maler [5], pp. 140–156, DOI: 10.1007/978-3-642-02658-4_14.
- [3] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2012): *Aca-cia+, a Tool for LTL Synthesis*. In: CAV, LNCS 7358, Springer, pp. 652–657, DOI: 10.1007/978-3-642-31424-7_45.

- [4] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot & Jean-François Raskin (2013): *Synthesis from LTL Specifications with Mean-Payoff Objectives*. In Nir Piterman & Scott A. Smolka, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Lecture Notes in Computer Science 7795*, Springer, pp. 169–184, DOI: 10.1007/978-3-642-36742-7_12.
- [5] Ahmed Bouajjani & Oded Maler, editors (2009): *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science 5643*, Springer, DOI: 10.1007/978-3-642-02658-4.
- [6] Patricia Bouyer, Ulrich Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey & Jiri Srba (2008): *Infinite Runs in Weighted Timed Automata with Energy Constraints*. In: *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, pp. 33–47, DOI: 10.1007/978-3-540-85778-5_4.
- [7] Lubos Brim, Jakub Chaloupka, Laurent Doyen, Raffaella Gentilini & Jean-François Raskin (2011): *Faster algorithms for mean-payoff games*. *Formal Methods in System Design* 38(2), pp. 97–118, DOI: 10.1007/s10703-010-0105-x.
- [8] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. *IEEE Trans. Computers* 35(8), pp. 677–691, DOI: 10.1109/TC.1986.1676819.
- [9] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger & Jean-François Raskin (2010): *Generalized Mean-payoff and Energy Games*. In Kamal Lodaya & Meena Mahajan, editors: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India, LIPIcs 8*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 505–516, DOI: 10.4230/LIPIcs.FSTTCS.2010.505. Available at <http://drops.dagstuhl.de/opus/portals/extern/index.php?seminr=10007>.
- [10] Krishnendu Chatterjee, Mickael Randour & Jean-François Raskin (2014): *Strategy synthesis for multi-dimensional quantitative objectives*. *Acta Inf.* 51(3-4), pp. 129–163, DOI: 10.1007/s00236-013-0182-6.
- [11] J. Crampton & G. Loizou (2001): *The completion of a poset in a lattice of antichains*. *International Mathematical Journal* 1(3), pp. 223–238. Available at <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.135.5671>.
- [12] Laurent Doyen & Jean-François Raskin (2010): *Antichain Algorithms for Finite Automata*. In Javier Esparza & Rupak Majumdar, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, Lecture Notes in Computer Science 6015*, Springer, pp. 2–22, DOI: 10.1007/978-3-642-12002-2_2.
- [13] A. Ehrenfeucht & J. Mycielski (1979): *Positional strategies for mean payoff games*. *International Journal of Game Theory* 8(2), pp. 109–113, DOI: 10.1007/BF01768705.
- [14] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2009): *An Antichain Algorithm for LTL Realizability*. In Bouajjani & Maler [5], pp. 263–277, DOI: 10.1007/978-3-642-02658-4_22.
- [15] Masahiro Fujita, Patrick C. McGeer & Jerry Chih-Yuan Yang (1997): *Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation*. *Formal Methods in System Design* 10(2/3), pp. 149–169, DOI: 10.1023/A:1008647823331.
- [16] Michael Huth & Mark Dermot Ryan (2004): *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, DOI: 10.1017/CBO9780511810275.
- [17] Shahar Maoz & Jan Oliver Ringert (2015): *GR(1) synthesis for LTL specification patterns*. In: *ESEC/FSE, ACM*, pp. 96–106, DOI: 10.1145/2786805.2786824.

- [18] Shahar Maoz & Jan Oliver Ringert (2015): *Synthesizing a Lego Forklift Controller in GR(1): A Case Study*. In: *Proc. 4th Workshop on Synthesis, SYNT 2015 colocated with CAV 2015, EPTCS 202*, pp. 58–72, DOI: 10.4204/EPTCS.202.5.
- [19] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, ACM Press, pp. 179–190, DOI: 10.1145/75277.75293.
- [20] Amir Pnueli, Yaniv Sa’ar & Lenore D. Zuck (2010): *JTLV: A Framework for Developing Verification Algorithms*. In: *CAV, LNCS 6174*, Springer, pp. 171–174, DOI: 10.1007/978-3-642-14295-6_18.